

FIG. 1

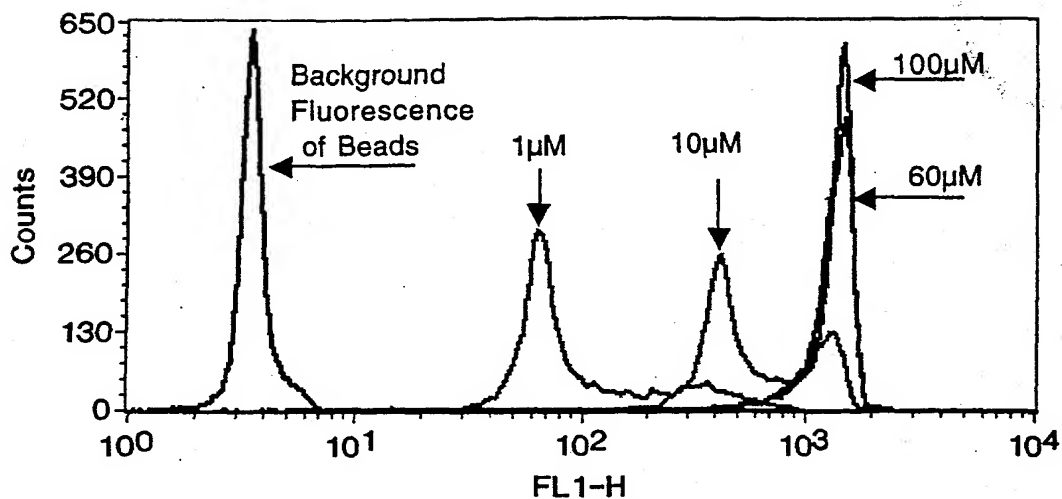


FIG. 2

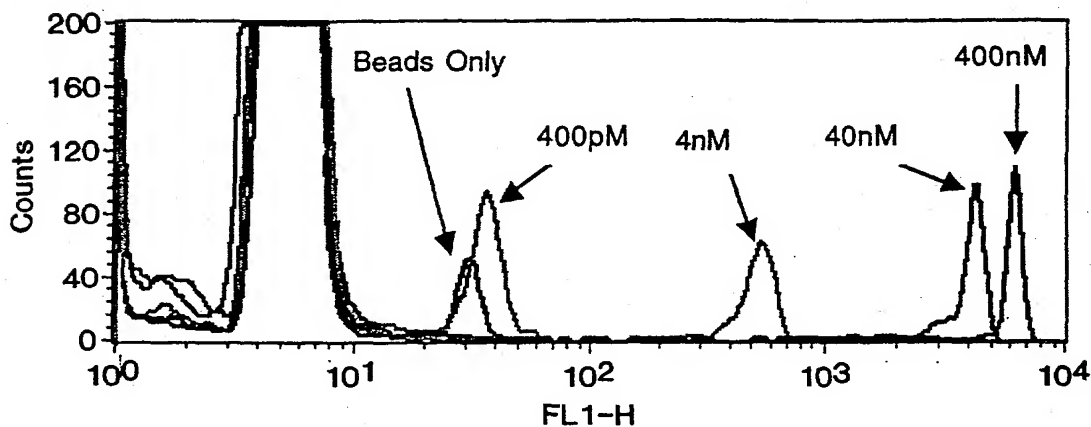


FIG. 3

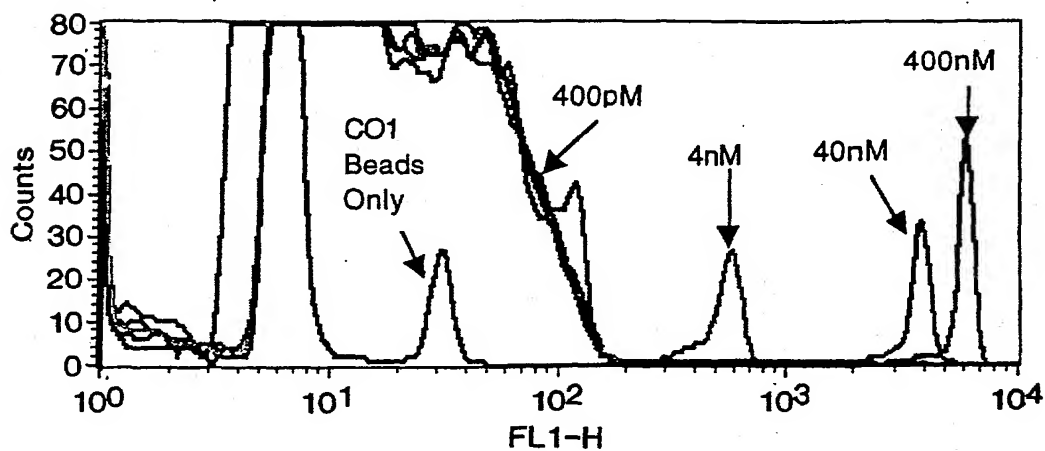


FIG. 4

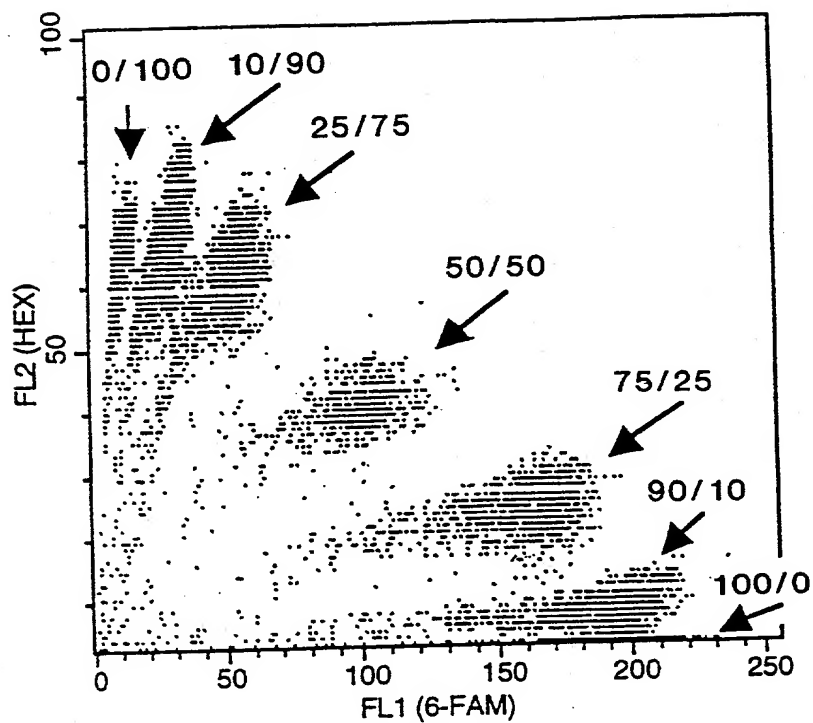


FIG. 5A

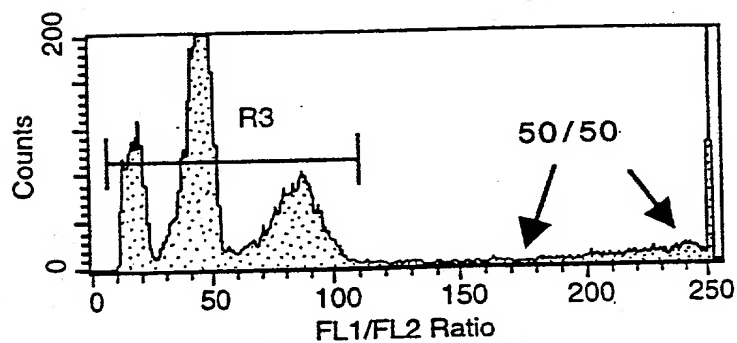


FIG. 5B-1

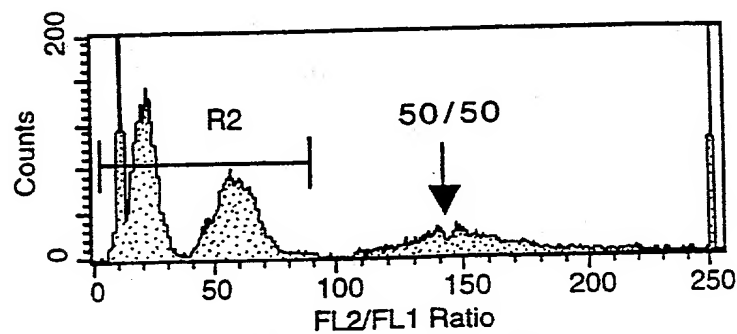


FIG. 5B-2

203220" 99E500T

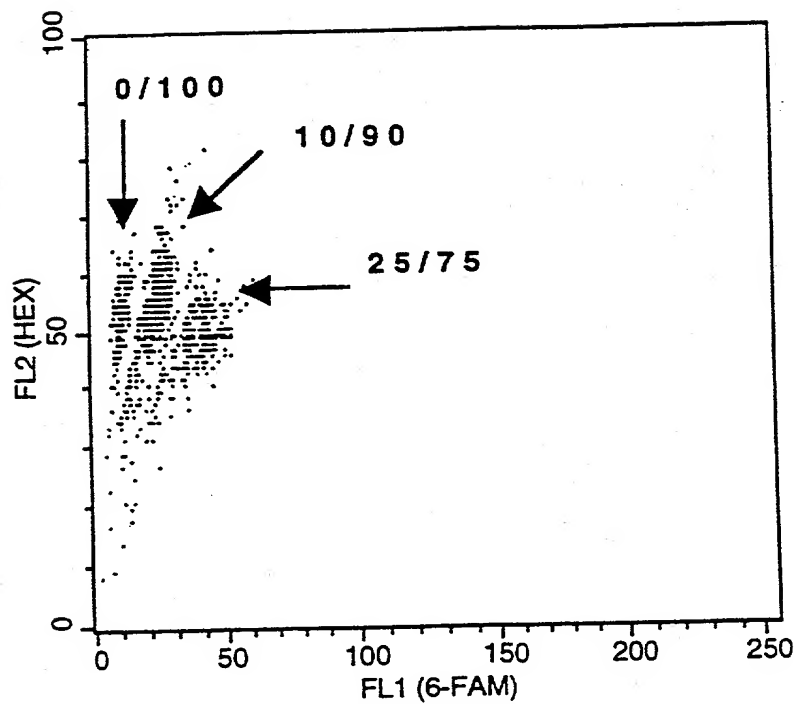


FIG. 5C

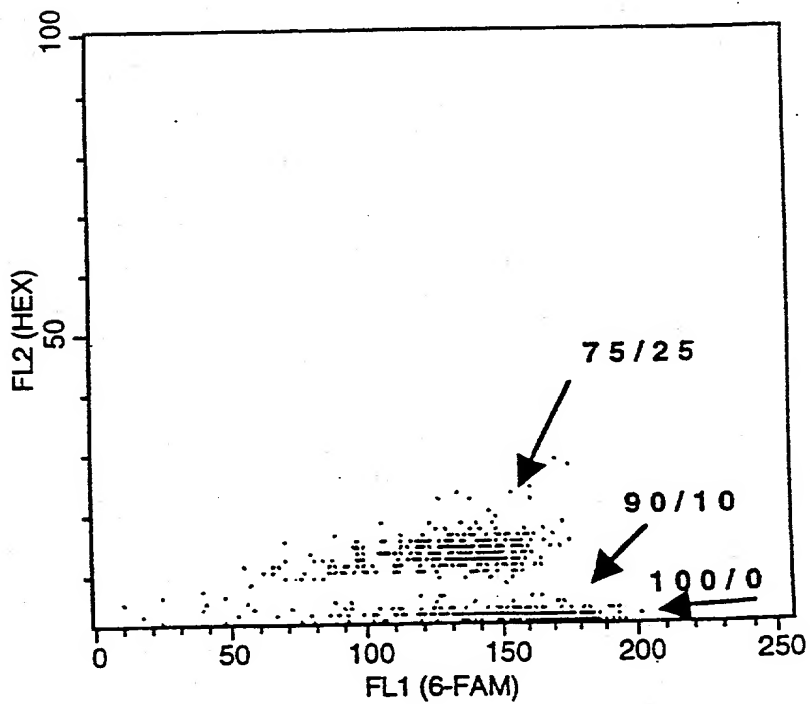
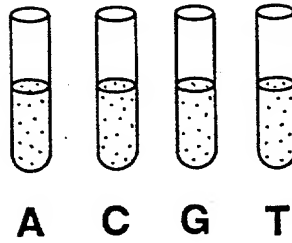


FIG. 5D

*Random N-mers*

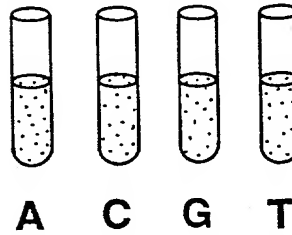
1. Couple to beads



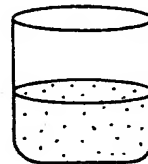
2. Pool and split



3. Couple



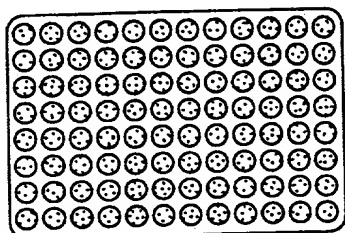
4. Repeat....



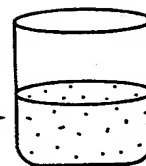
**FIG. 6**

**Sequence Identifier Tags**

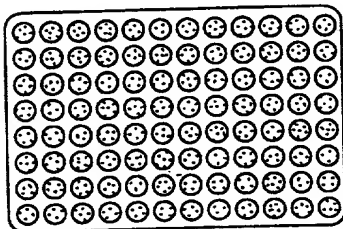
1. 8 synthetic coupling reactions



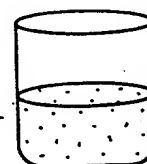
2. Pool and Split



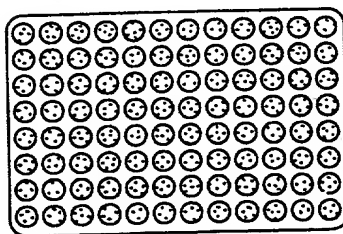
3. 8 further coupling reactions



4. Pool and Split



5. 8 further coupling reactions



6. Final product: one million 24-mers

**FIG. 7**

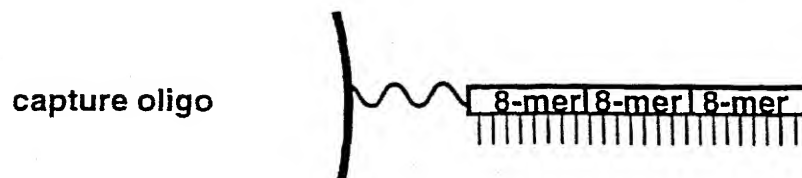
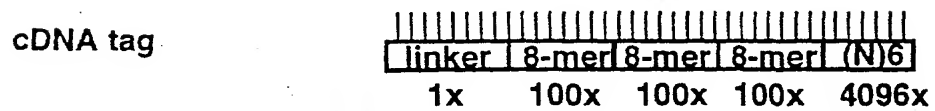
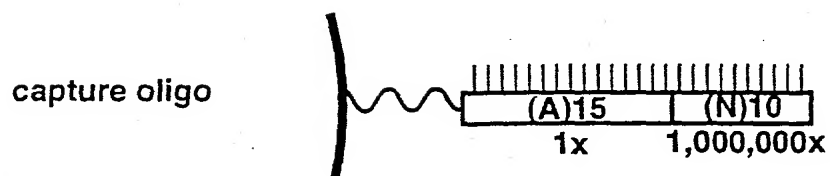
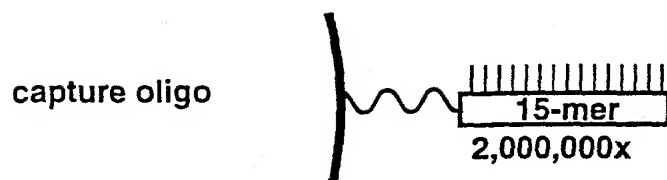
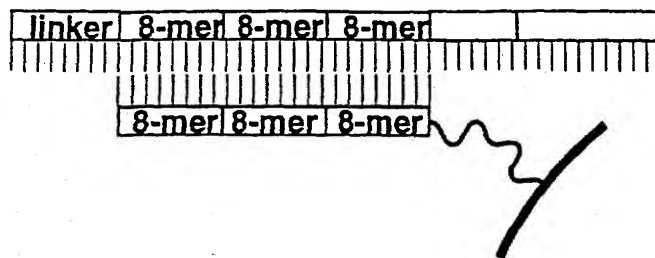


FIG. 8

Perfect match

predicted T<sub>m</sub>: 72 deg.



Mismatches

predicted T<sub>m</sub>: <48 deg.

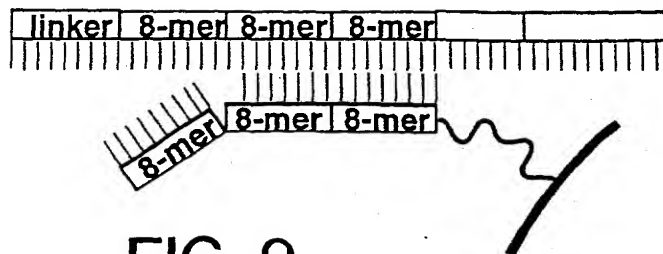
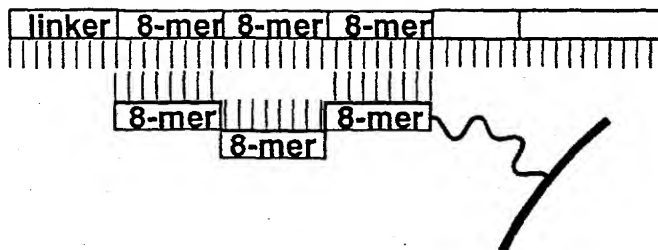
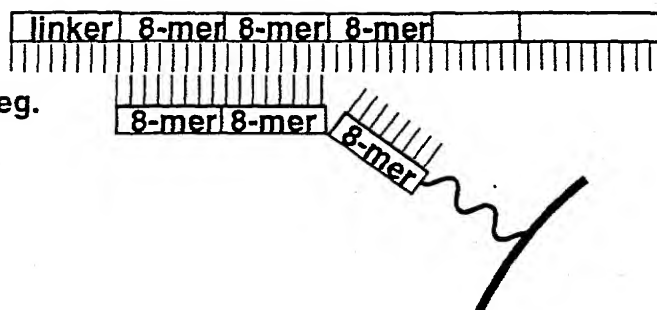
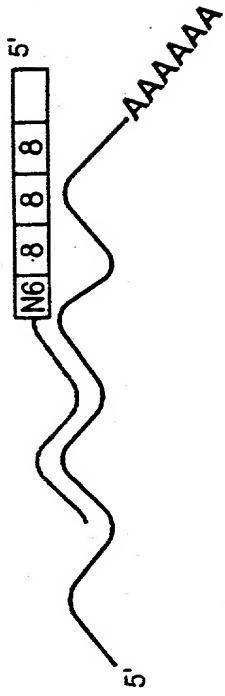


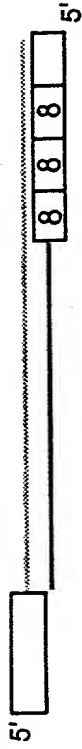
FIG. 9



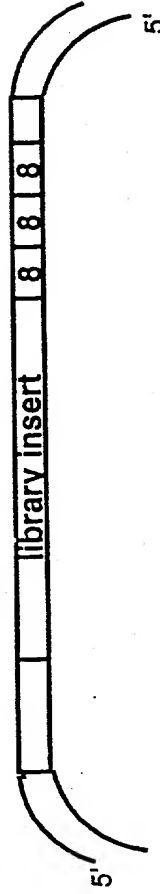
## 1. first strand synthesis



## 2. second strand synthesis



### 3. polish ends and clone



**FIG. 10**

Prepare  
labeled cDNA

Hybridize  
with beads

Sort with  
FACS

FAM < HEX

Waste

FAM > HEX

PCR

Reintroduction  
into cells

DNA sequence  
analysis

FIG. 11

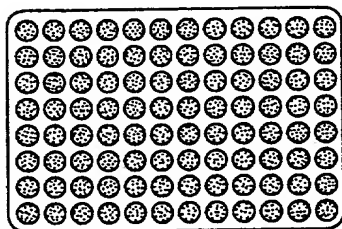
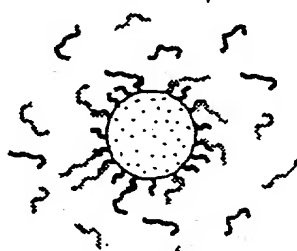
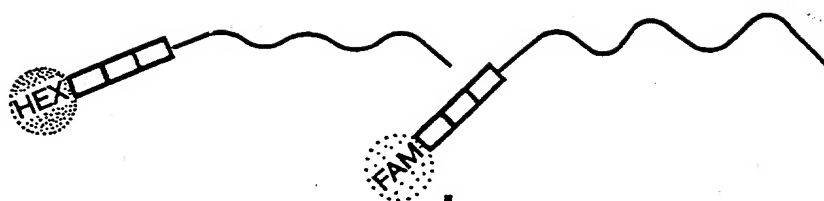


FIG. 12

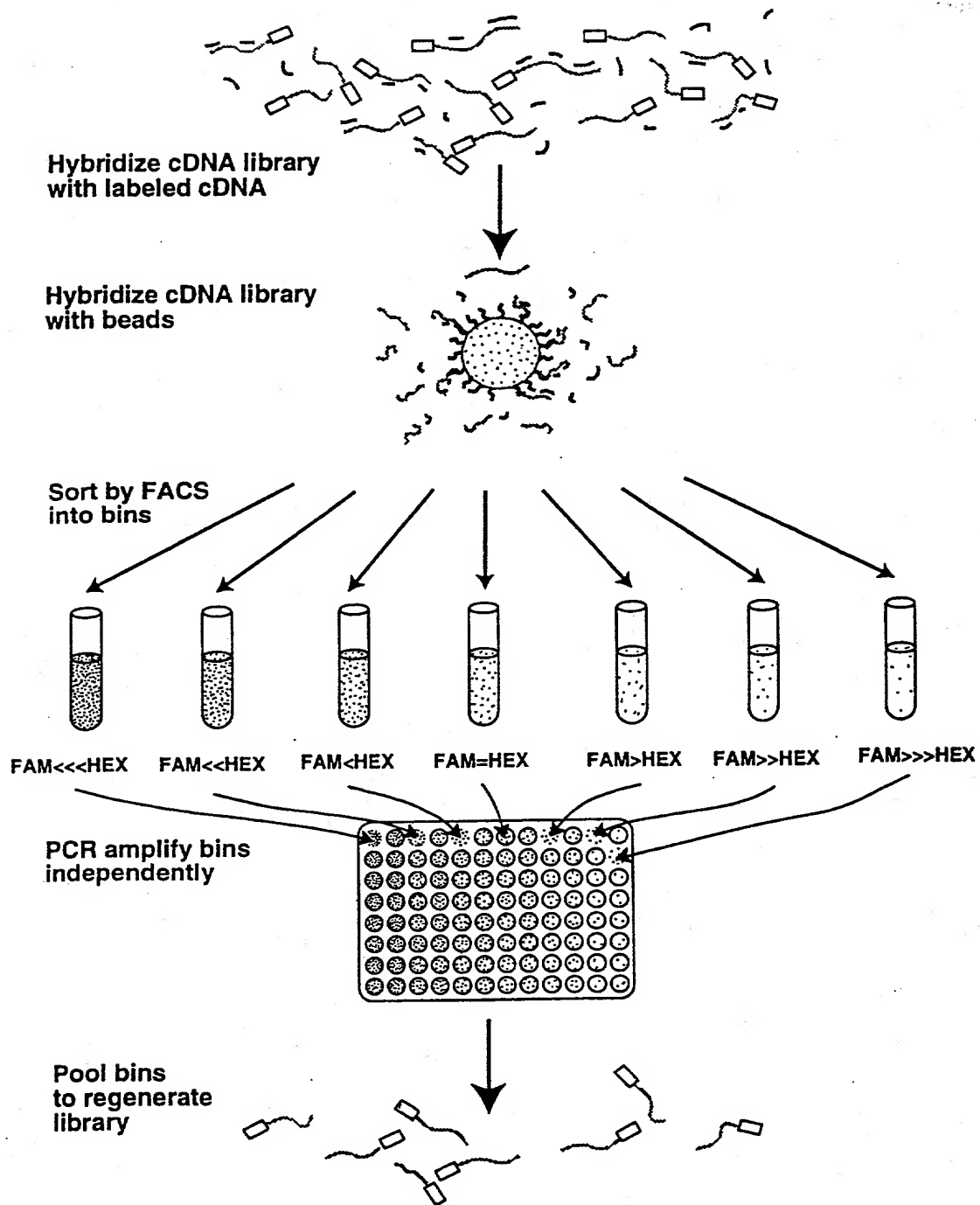


FIG. 13

Hybridize labeled  
cellular cDNA with  
unlabeled library cDNA

Hybridize  
with beads

Sort with  
FACS

FAM < HEX

Waste

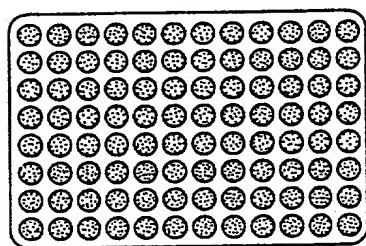
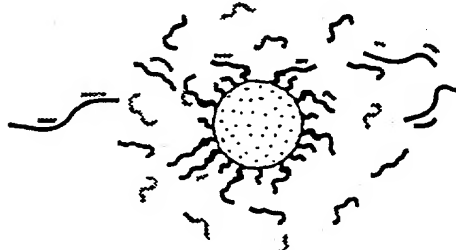
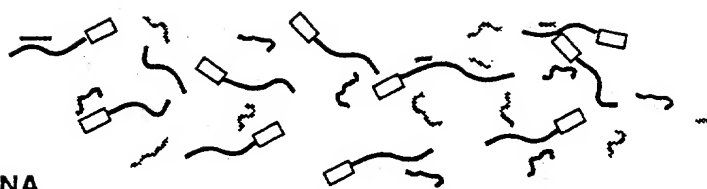
FAM > HEX

PCR

Reintroduction  
into cells

DNA sequence  
analysis

FIG. 14



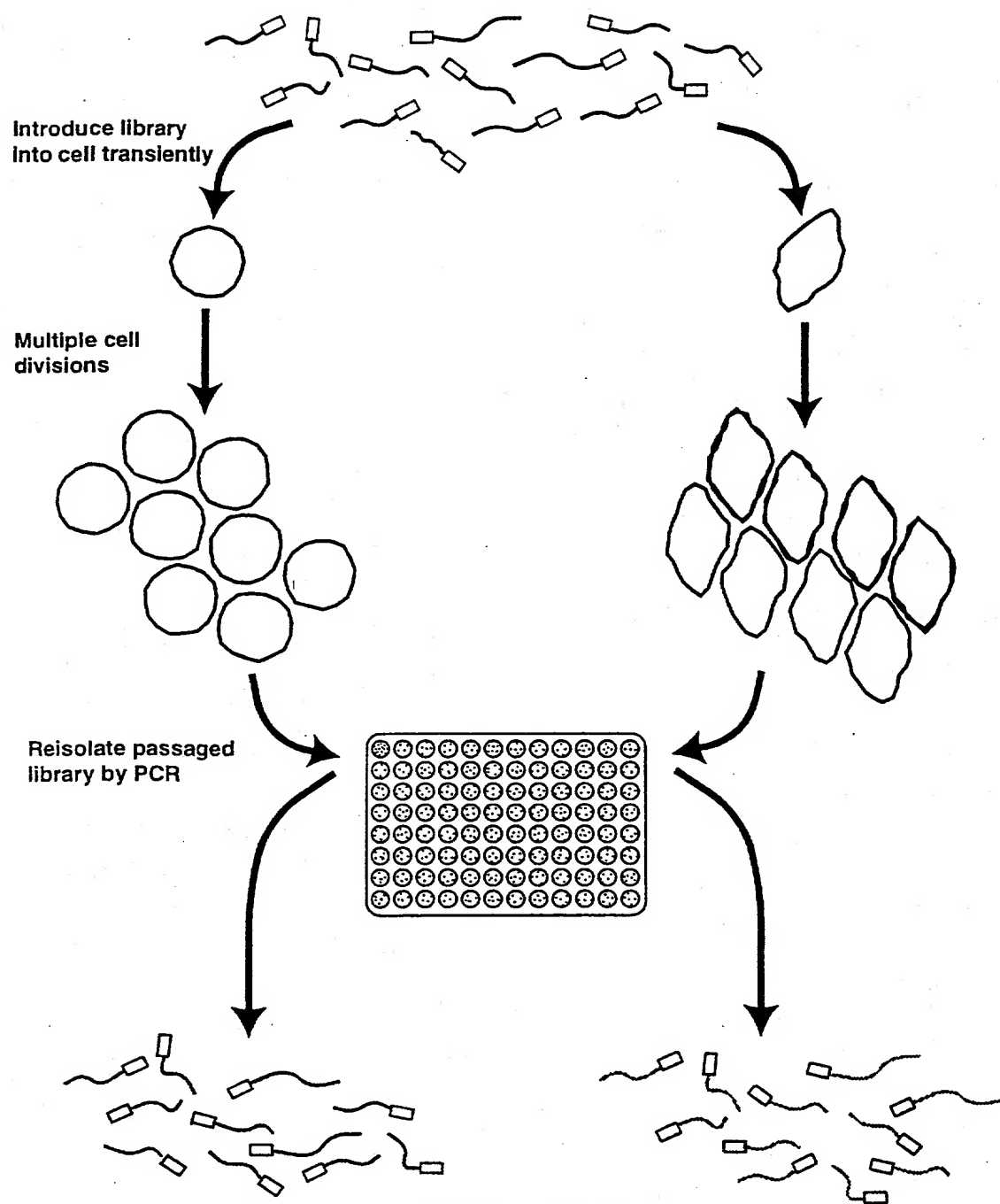


FIG. 15A

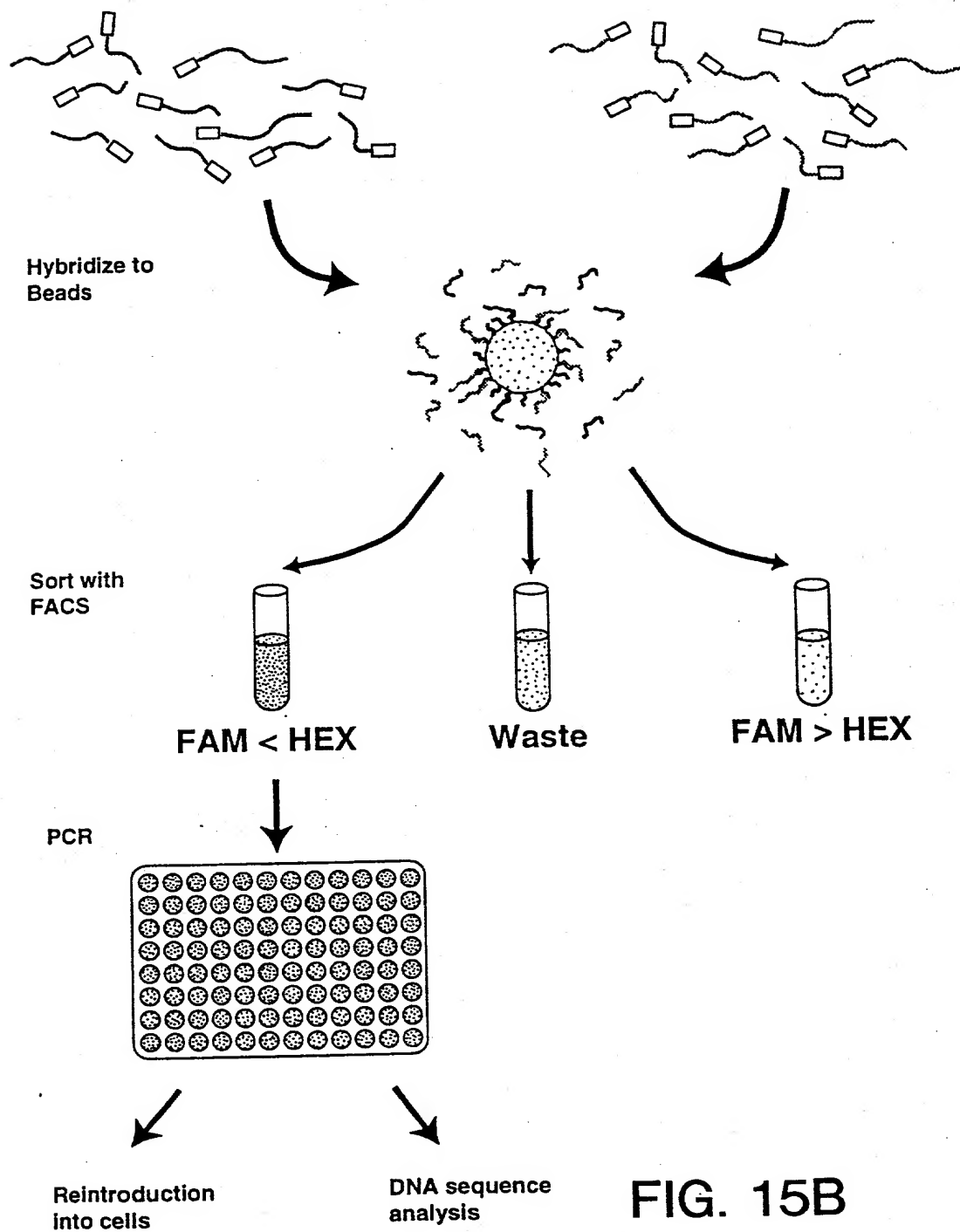


FIG. 15B

```

#include <stdio.h>

#define MATCH_NEEDED_IN_2ND 3
#define LEN_MER 8 // recompile when changed.  *things to pay attention to.

int SecondStruct(const char*);

int CrossHyb(const char *str1, const char *str2, int overlap_length);
int SimpleMatch(const char *str1, const char *str2);
char FirstN(const char *str1, const char *str2, int N);

int GC_ADDITION = 1;
int NUM_GC = 4;
int SCORE_NEEDED_IN_2ND = 7; // 1+2+4
char eq1, eq2;

FILE *fp = stdout;
FILE *fplog = stderr;

main(int argc, char **argv)
{
    int ii, jj, comp_score, s;
    int MM[LEN_MER];
    char line[256], str[LEN_MER+1];
    int gcsum, pass_gc, total_probes, failed_fn, failed_ch, failed_sm;
    char convert[4]; // 0123 to atcg conversion.
    char *compatible;

    int max_prb, cnt_prb;
    char **probe;

    int max_snd, cnt_snd;
    char **sndstr;
    int *snd_matchcnt;

    int SIMPLE_CUTOFF = 5; // reject if this many bps match to each other,
                          // no matter where they are located.
    int CROSSHYB_CUTOFF = 9; // 1+2+4 + 2
    int CROSSHYB_OVERLAP = 5;

    int FIRST_N = 4;

    eq1 = eq2 = '';
    if(argc == 1)
    {

```

**FIG. 16A**



```

fprintf(stderr,
    "Usage: %s -o output_file[stdout]\n",
    argv[0]);
fprintf(stderr,
    "\t\t-gc number_of_GC_in_probe[%d]\n",
    NUM_GC);
fprintf(stderr,
    "\t\t-2 secondary_structure_reject(including this value)[%d]\n",
    SCORE_NEEDED_IN_2ND);
fprintf(stderr,
    "\t\t-ch crosshyb_reject(including this value)[%d]\n",
    CROSSHYB_CUTOFF);
fprintf(stderr,
    "\t\t-sm simple_match_reject(including this value)[%d]\n",
    SIMPLE_CUTOFF);
fprintf(stderr,
    "\t\t-ol crosshyb_overlap_length[%d]\n",
    CROSSHYB_OVERLAP);
fprintf(stderr,
    "\t\t-eq add'l_equiv_bp_in_compat_checking[%c%c] (e.g., -eq gt)\n",
    eq1, eq2);
fprintf(stderr, "\t\t-fn first_N_length[%d]\n", FIRST_N);
fprintf(stderr, "\t\t-gc_add GC_add'l_penalty[%d]\n", GC_ADDITION);

exit(0);
}

// parse input parameters.
ii = 1;
while(ii < argc)
{
    if(strcmp(argv[ii], "-gc") == 0)
        sscanf(argv[ii+1], "%d", &NUM_GC);

    else if(strcmp(argv[ii], "-2") == 0)
        sscanf(argv[ii+1], "%d", &SCORE_NEEDED_IN_2ND);

    else if(strcmp(argv[ii], "-ch") == 0)
        sscanf(argv[ii+1], "%d", &CROSSHYB_CUTOFF);

    else if(strcmp(argv[ii], "-ol") == 0)
        sscanf(argv[ii+1], "%d", &CROSSHYB_OVERLAP);

    else if(strcmp(argv[ii], "-eq") == 0)
    {

```

**FIG. 16B**

```

if(strlen(argv[ii+1]) == 2 ||
   (strlen(argv[ii+1]) == 3 && argv[ii+1][2] == '\n'))
{
    eq1 = argv[ii+1][0];
    eq2 = argv[ii+1][1];
}
else
{
    fprintf(stderr, "\nERROR: Invalid string after -eq flag.\n\n");
    exit(1);
}
}

else if(strcmp(argv[ii], "-o") == 0)
{
    if((fp = fopen(argv[ii+1], "w")) == NULL)
    {
        fprintf(stderr, "Can't open file %s to write.\n", argv[ii+1]);
        exit(1);
    }

    char logname[128];
    sprintf(logname, "%s.log", argv[ii+1]);
    if((fplog = fopen(logname, "w")) == NULL)
    {
        fprintf(stderr, "failed creating log. stderr used.\n");
        fplog = stderr;
    }
}

else if(strcmp(argv[ii], "-fn") == 0)
{
    sscanf(argv[ii+1], "%d", &FIRST_N);
}

else if(strcmp(argv[ii], "-sm") == 0)
{
    sscanf(argv[ii+1], "%d", &SIMPLE_CUTOFF);
}

else if(strcmp(argv[ii], "-gc_add") == 0)
{
    sscanf(argv[ii+1], "%d", &GC_ADDITION);
}

```

FIG. 16C

```

else
{
    fprintf(stderr, "Unknow flag %s\n", argv[ii]);
    exit(1);
}

ii += 2;
}

max_prb = 30000;
probe = new char* [max_prb];
for(ii = 0; ii < max_prb; ii++)
    probe[ii] = new char [LEN_MER+1];
cnt_prb = 0;

max_snd = 5000;
sndstr = new char* [max_snd];
for(ii = 0; ii < max_snd; ii++)
    sndstr[ii] = new char [LEN_MER+1];
snd_matchcnt = new int [max_snd];
cnt_snd = 0;

// build an array of probes. Each probe is of length LEN_MER,
// of which 'NUM_GC' are Gs or Cs.

convert[0] = 'a';
convert[1] = 't';
convert[2] = 'c';
convert[3] = 'g';

total_probes = 0;
pass_gc = 0; // number of probes pass GC test.
for(MM[0] = 0; MM[0] < 4; MM[0]++)
for(MM[1] = 0; MM[1] < 4; MM[1]++)
for(MM[2] = 0; MM[2] < 4; MM[2]++)
for(MM[3] = 0; MM[3] < 4; MM[3]++)
for(MM[4] = 0; MM[4] < 4; MM[4]++) //things to pay attention to.
for(MM[5] = 0; MM[5] < 4; MM[5]++) //things to pay attention to.
for(MM[6] = 0; MM[6] < 4; MM[6]++) //things to pay attention to.
for(MM[7] = 0; MM[7] < 4; MM[7]++) //things to pay attention to.
{
    total_probes++;
    gcsum = 0;

    // build a probe.

```

FIG. 16D

```

for(jj = 0; jj < LEN_MER; jj++)
{
    str[jj] = convert[MM[jj]];
    if(str[jj] == 'c' || str[jj] == 'g')
        gcsum++;
}
str[LEN_MER] = '\0';

// check its GC contents and secondary structure.
if(gcsum == NUM_GC)
{
    pass_gc++;

    fprintf(fplog, "pass GCtest: %s\n", str);

    if(!SecondStruct(str))
    {
        strcpy(probe[cnt_prb], str);
        if(++cnt_prb == max_prb)
        {
            // should relocate memory.
            // To simplify the program, let's just give an error msg.
            fprintf(stderr, "ERROR: Probe array is too small. cnt_prb is %d\n", cnt_prb);
            exit(1);
        }
    }
    else
    {
        // record the rejected string
        strcpy(sndstr[cnt_snd], str);
        if(++cnt_snd == max_snd)
        {
            fprintf(stderr, "ERROR: Secondary Structure array is too small. cnt_snd = %d\n",
cnt_snd);
            exit(1);
        }
    }
}

fprintf(fp, "\n%d mer probe selection\n", LEN_MER);
fprintf(fp, "Number of GCs in the probes: %d\n", NUM_GC);
fprintf(fp, "Score to reject as secondary structure: %d\n",
SCORE_NEEDED_IN_2ND);
fprintf(fp, "Score to reject as incompatible: %d\n", CROSSHYB_CUTOFF);

```

**FIG. 16E**

```

fprintf(fp, "Compatible test overlap: %d\n", CROSSHYB_OVERLAP);
fprintf(fp, "Additional equivalent base-pair in compatibility checking: %c%c\n",
        eq1, eq2);
fprintf(fp, "Simple match cutoff value(including): %d\n", SIMPLE_CUTOFF);
fprintf(fp, "First N value(including): %d\n", FIRST_N);
fprintf(fp, "Additional penalty for G or C: %d\n", GC_ADDITION);
fprintf(fp, "\n\n");

fprintf(fp, "Total possible %d mers: %d\n", LEN_MER, total_probes);
fprintf(fp, "Number passed GC_test : %d\n", pass_gc);
fprintf(fp, "Number passed secondary structure test : %d\n", cnt_prb);
// for(ii = 0; ii < cnt_snd; ii++)
//     fprintf(fp, "%s\n", sndstr[ii]);

// From the set (call it set1) of probes which passed GC and 2nd structure
// tests, choose a probe into the final set(set2). Then compare this
// probe against all the probes left in set1 and thought out the ones
// that may crosshyb to this probe. From what's left in set1, choose
// another probe and compary it to the rest of set1...

compatible = new char [cnt_prb];
for(ii = 0; ii < cnt_prb; ii++)
{
    compatible[ii] = 'T';
}

// Compatibility check #1: Use weighted scores to penalize neighboring matches.
// first_match_score = 1;
// if prev pair is a match, current_match_score = prev_match_score*2.
ii = 0;
failed_ch = 0;
while(ii < cnt_prb)
{
    for(jj = ii+1; jj < cnt_prb; jj++)
    {
        if(compatible[jj] == 'T' &&
           (s=CrossHyb(probe[ii],probe[jj],CROSSHYB_OVERLAP)) >= CROSSHYB_CUTOFF)
        {
            compatible[jj] = 'F';
            failed_ch++;
            fprintf(fplog, "Rejected(%d) %s in slide test for %s\n",
                    s, probe[jj], probe[ii]);
        }
    }
}

```

**FIG. 16F**

```

    ii++;
    while(ii < cnt_prb && compatible[ii] == 'F')
        ii++;
}
fprintf(fp, "Number of probes passed compatibility test: %d\n",
        cnt_prb - failed_ch);

// Compatibility check #2: Use unweighted score: count unconsecutive matches

// find the first 'passed' probe.
ii = 0;
while(ii < cnt_prb && compatible[ii] == 'F')
    ii++;
failed_sm = 0;
while(ii < cnt_prb)
{
    for(jj = ii+1; jj < cnt_prb; jj++)
    {
        if(compatible[jj] == 'T' &&
            (s=SimpleMatch(probe[ii],probe[jj])) >= SIMPLE_CUTOFF)
        {
            compatible[jj] = 'F';
            fprintf(fplog, "Rejected(%d) %s in simple_match test for %s\n",
                    s, probe[jj], probe[ii]);
            failed_sm++;
        }
    }

    ii++;
    while(ii < cnt_prb && compatible[ii] == 'F')
        ii++;
}
fprintf(fp, "Number of probes passed simple match test: %d\n",
        cnt_prb - failed_ch - failed_sm);

// Compatibility check #3: if the first N bases match ANYWHERE in another probe.

// find the first 'passed' probe.
ii = 0;
while(ii < cnt_prb && compatible[ii] == 'F')
    ii++;
failed_fn = 0;
while(ii < cnt_prb)

```

FIG. 16G

```

{
    for(jj = ii+1; jj < cnt_prb; jj++)
    {
        if(compatible[jj] == 'T' &&
           FirstN(probe[ii], probe[jj], FIRST_N) == 'T')
        {
            compatible[jj] = 'F';
            failed_fn++;
            fprintf(fplog, "Rejected %s in FIRSTN test for %s\n",
                    probe[jj], probe[ii]);
        }
    }

    ii++;
    while(ii < cnt_prb && compatible[ii] == 'F')
        ii++;
}

fprintf(fp, "Number of probes passed FIRSTN compatibility test: %d\n",
        cnt_prb - failed_ch - failed_sm - failed_fn);

// output.
jj = 0;
fprintf(fp, "\nSelected probes are: \n");
for(ii = 0; ii < cnt_prb; ii++)
{
    if(compatible[ii] == 'T')
    {
        fprintf(fp, "%s \n", probe[ii]);
        jj++;
    }
}

}

// Check if 'str' contains a secondary structure. That is, if there is a
// consecutive 3 bases that matches when 'str' is folded.
// return 1 if found secondary structure, 0 otherwise.

int SecondStruct(const char *str)
{
    int ii, jj, kk, ll;
    int sum, score[32];
    char prev_match;
    char *compl;

```

FIG. 16H

```

char complement[256];
complement['a'] = 't';
complement['t'] = 'a';
complement['c'] = 'g';
complement['g'] = 'c';

```

```

ll = strlen(str);
compl = new char [ll+1];
for(ii = 0; ii < ll; ii++)
{
    compl[ii] = complement[str[ii]];
}

```

```

for(ii = MATCH_NEEDED_IN_2ND; ii < ll - MATCH_NEEDED_IN_2ND; ii++)
{
    prev_match = 'F';
    sum = 0;
    for(jj = 0; jj < ii; jj++)
    {
        score[jj] = 0;
        kk = ii*2 - jj;
        if(kk < ll)
        {
            if(str[jj] == compl[kk])
            {
                if(prev_match == 'T')
                {
                    score[jj] = score[jj-1] * 2;
                }
                else
                {
                    score[jj] = 1;
                    prev_match = 'T';
                }
            }
            else
            {
                prev_match = 'F';
            }
        }
        sum += score[jj];
    }
}

```

```

// fprintf(stderr, "2' sum = %d\n", sum);
if(sum >= SCORE_NEEDED_IN_2ND)

```

**FIG. 16I**



```

    {
        delete [] compl;
        return 1; // Found a 2nd structure.
    }
}

for(ii = MATCH_NEEDED_IN_2ND - 1; ii < ll - MATCH_NEEDED_IN_2ND; ii++)
{
    prev_match = 'F';
    sum = 0;
    for(jj = 0; jj <= ii; jj++)
    {
        score[jj] = 0;
        kk = ii*2+1 - jj;
        if(kk < ll)
        {
            if(str[jj] == compl[kk])
            {
                if(prev_match == 'T')
                {
                    score[jj] = score[jj-1]*2;
                }
                else
                {
                    score[jj] = 1;
                    prev_match = 'T';
                }
            }
            else
            {
                prev_match = 'F';
            }
        }
        sum += score[jj];
    }

    // fprintf(stderr, "2' sum = %d\n", sum);
    if(sum >= SCORE_NEEDED_IN_2ND)
    {
        delete [] compl;
        return 1; // Found a 2nd structure.
    }
}

delete [] compl;

```

FIG. 16J

```

    return 0; // No 2nd structure.
}

// check if str1 and str2 can hybridize together.
// return the max of match scores.
// Assume strlen(str1) == strlen(str2).

int CrossHyb(const char *str1, const char *str2, int overlap)
{
    int ii, jj, len, sum, score, prev_score, max_sum, numGC;
    char prev_match;

    len = strlen(str1);
    max_sum = 0;

    fprintf(fplog, "Sliding test between %s and %s\n", str1, str2);

    for(ii = overlap-len; ii <= len-overlap; ii++)
    {
        numGC = 0;
        sum = 0;
        score = prev_score = 0;
        prev_match = 'F';
        fprintf(fplog, "Compare ");
        for(jj = ii; jj < len && jj - ii < len; jj++)
        {
            if(jj >= 0 && jj - ii >= 0)
            {
                fprintf(fplog, "(%c,%c) ", str1[jj], str2[jj-ii]);
                if((str1[jj] == str2[jj-ii]) ||
                   (str1[jj] == eq1 && str2[jj-ii] == eq2) ||
                   (str1[jj] == eq2 && str2[jj-ii] == eq1))
                {
                    if(((str1[jj]>32) == 'g' && (str2[jj-ii]>32) == 'g') ||
                       ((str1[jj]>32) == 'c' && (str2[jj-ii]>32) == 'c'))
                        numGC++;

                    if(prev_match == 'T')
                    {
                        score = prev_score*2;
                    }
                    else
                    {
                        score = 1;
                    }
                }
            }
        }
    }
}

```

FIG. 16K

```

        prev_match = 'T';
    }
}
else
{
    score = 0;
    prev_match = 'F';
}
sum += score;
prev_score = score;
}

fprintf(fplog, "Score=%d\n", sum + numGC*GC_ADDITION);

if(sum + numGC*GC_ADDITION > max_sum)
    max_sum = sum + numGC*GC_ADDITION;
}

fprintf(fplog, "Max score is %d\n", max_sum);
return max_sum;
}

// Compare 2 strings base to base, 0 to 0, 1 to 1..., no sliding.
// return number of matches.
// Assume strlen(str1) == strlen(str2).

int SimpleMatch(const char *str1, const char *str2)
{
    int ii, sum;

    sum = 0;
    for(ii = 0; ii < strlen(str1); ii++)
    {
        if((str1[ii] == str2[ii]) ||
           (str1[ii] == eq1 && str2[ii] == eq2) ||
           (str1[ii] == eq2 && str2[ii] == eq1))
        {
            sum++;
        }
    }

    return sum;
}

```

FIG. 16L

```

// Check if the first N bases of the two probes are identical.
char FirstN(const char *str1, const char *str2, int N)
{
    int ii;
    char match = 'T';

    if(N > strlen(str1))
        return 'F';

    for(ii = 0; ii < N; ii++)
    {
        if(!((str1[ii] == str2[ii]) ||
            (str1[ii] == eq1 && str2[ii] == eq2) ||
            (str1[ii] == eq2 && str2[ii] == eq1))))
        {
            match = 'F';
            break;
        }
    }

    return match;
}

```

FIG. 16M

20320" 9965307

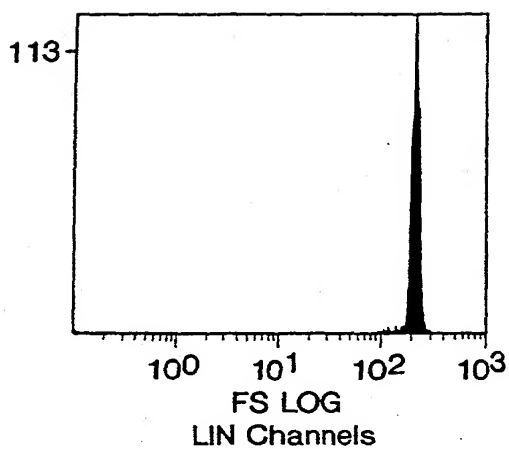


FIG. 17A-1

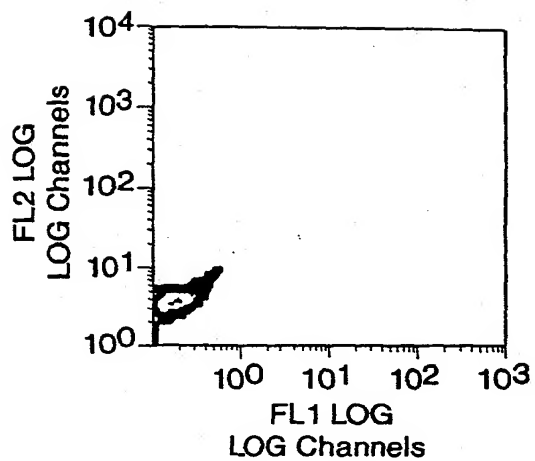


FIG. 17A-2

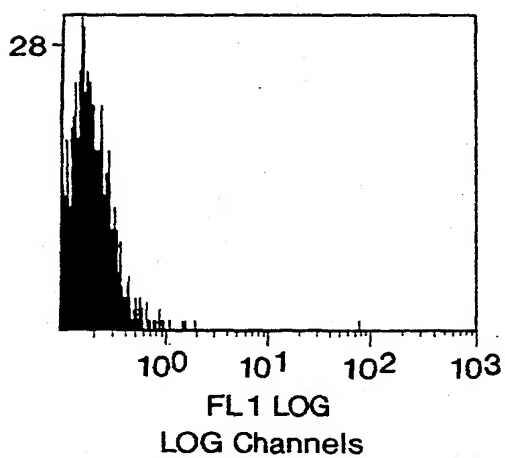


FIG. 17A-3

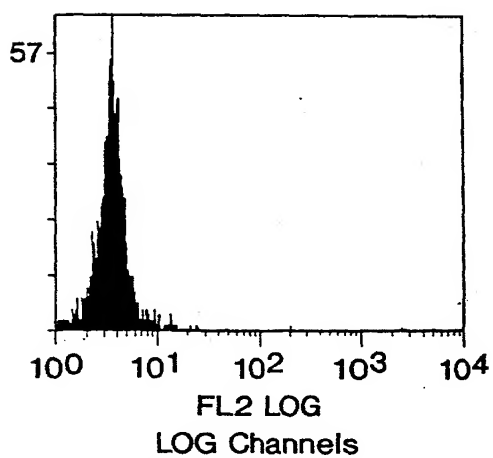


FIG. 17A-4

LIN-LOG  
Number  
2  
2599  
2492  
238

%Gated  
0.1  
86.6  
83.1  
7.9

X-Mean  
35.9  
1.8  
35.5  
661.2

Units(X axis)  
LOG Channels  
LOG Channels  
LOG Channels  
LOG Channels

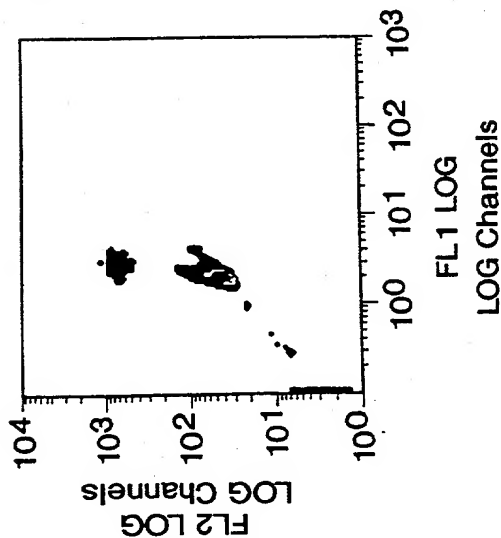


FIG. 17B-1

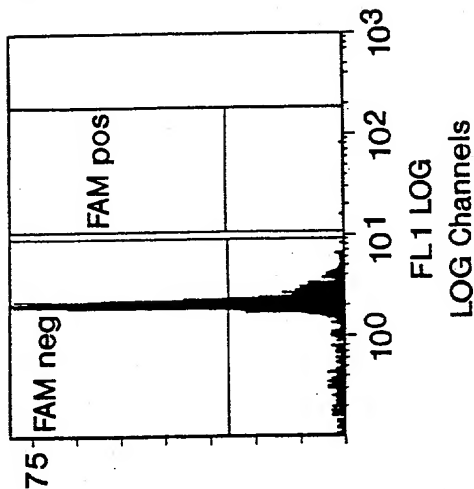


FIG. 17B-2

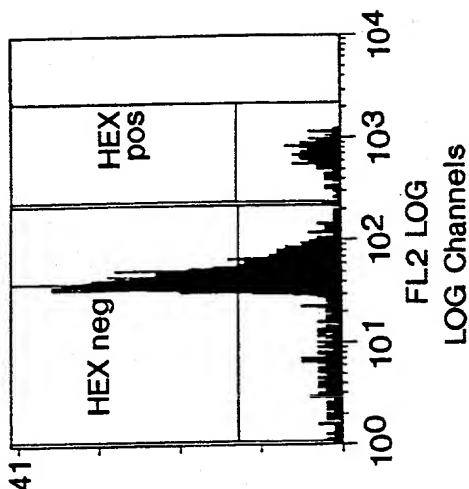
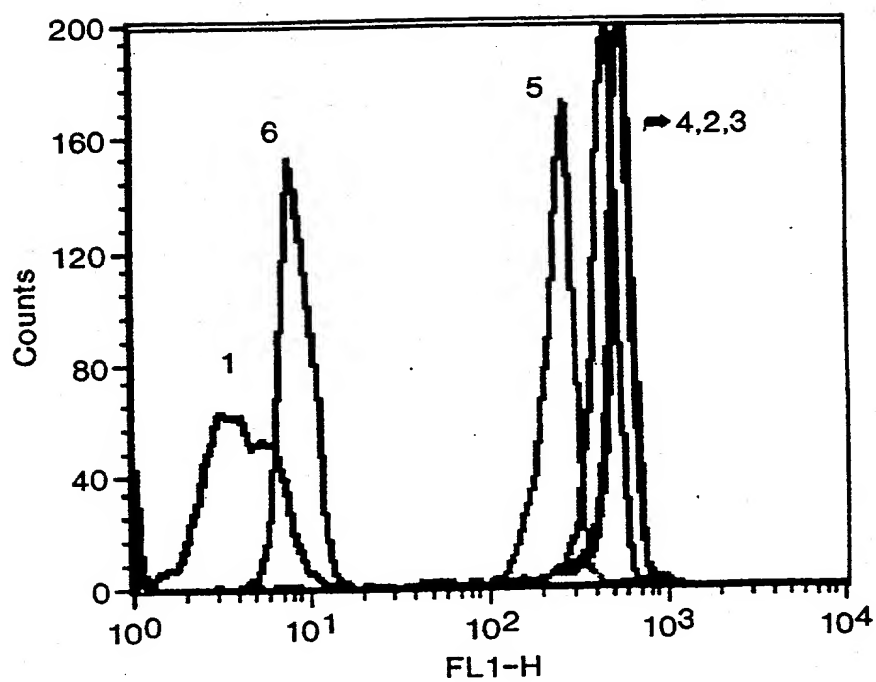
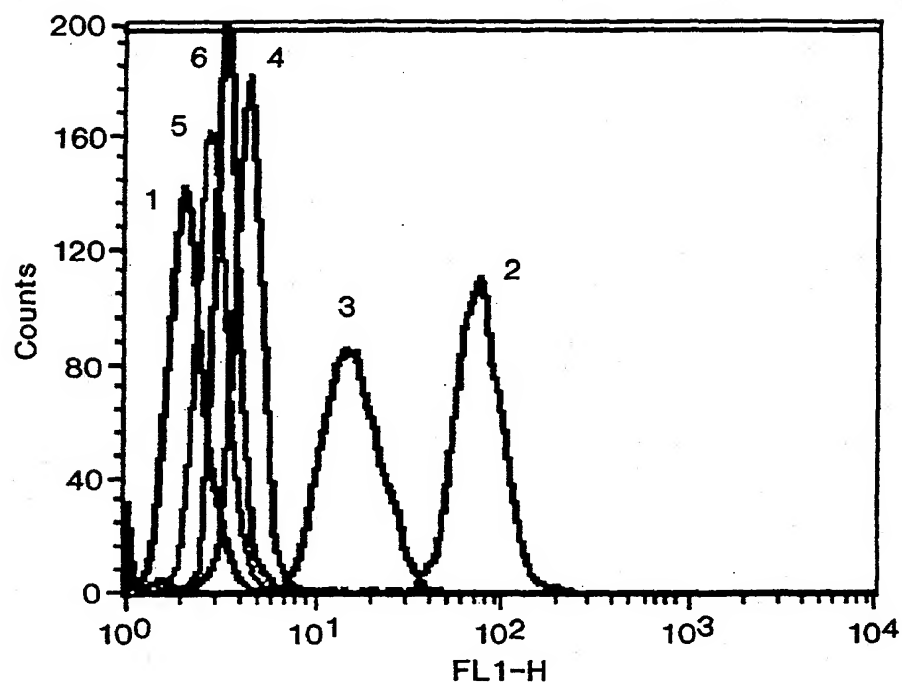


FIG. 17B-3



Key	Name
1	5' bead alone
2	5' bead/2 $\mu$ M 5' c'
3	5' bead/2 $\mu$ M 60mer DNA
4	5' bead/5 $\mu$ M 60mer RNA trans.
5	5' bead/1 $\mu$ M 60mer RNA trans.
6	5' bead/20 $\mu$ M Non-specific

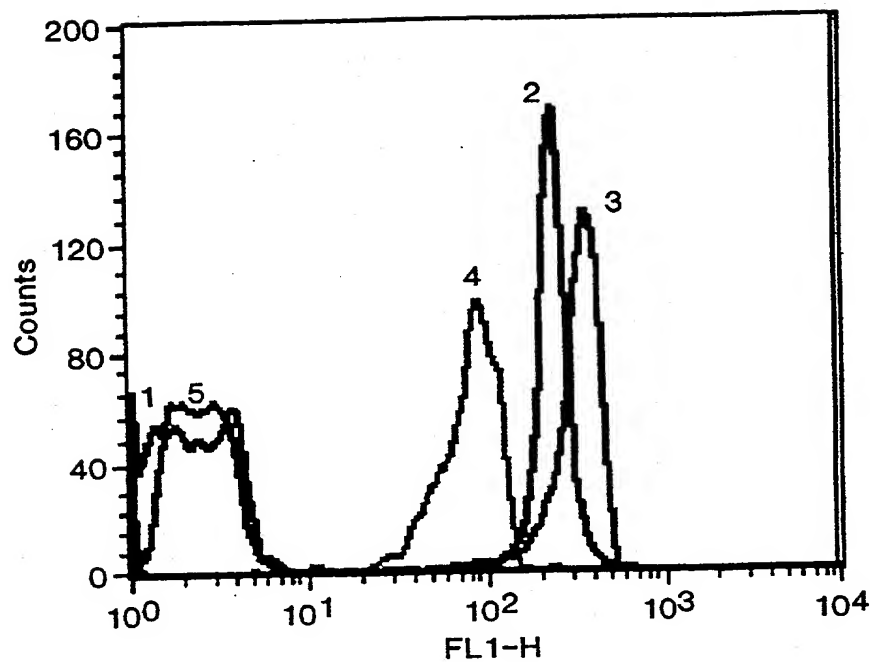
FIG. 18A



Key	Name
1	3' bead alone
2	3' bead/2 $\mu$ M 3' c'
3	3' bead/2 $\mu$ M 60mer DNA
4	3' bead/5 $\mu$ M 60mer RNA trans.
5	3' bead/1 $\mu$ M 60mer RNA trans.
6	3' bead/20 $\mu$ M Non-specific

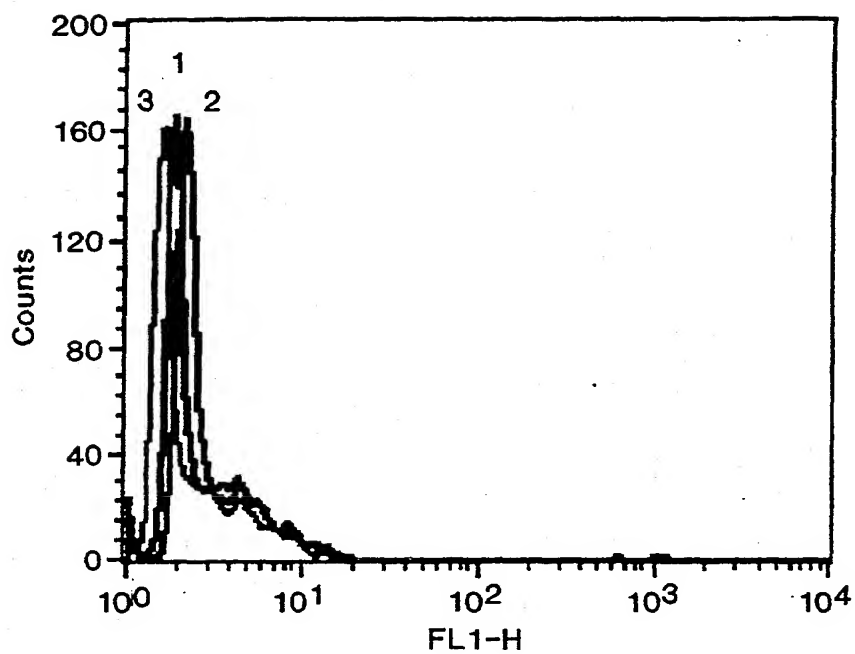
FIG. 18B





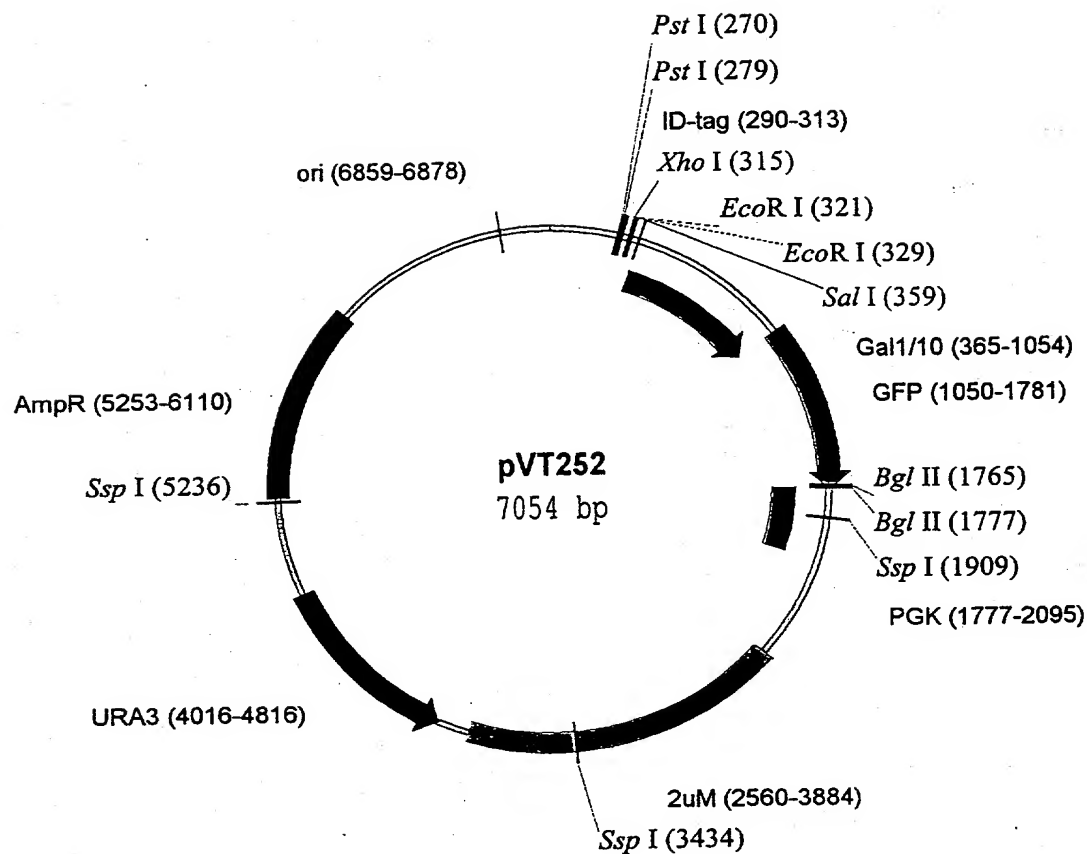
Key	Name
1	Mid bead alone
2	Mid bead/2 $\mu$ M 60mer DNA
3	Mid bead/5 $\mu$ M 60mer RNA trans.
4	Mid bead/1 $\mu$ M 60mer RNA trans.
5	Mid bead/20 $\mu$ M Non-specific

FIG. 18C



Key	Name
1	NS bead/2 $\mu$ M 60mer DNA
2	NS bead/5 $\mu$ M 60mer RNA trans.
3	NS bead/1 $\mu$ M 60mer RNA trans.

FIG. 18D



**FIG. 19**

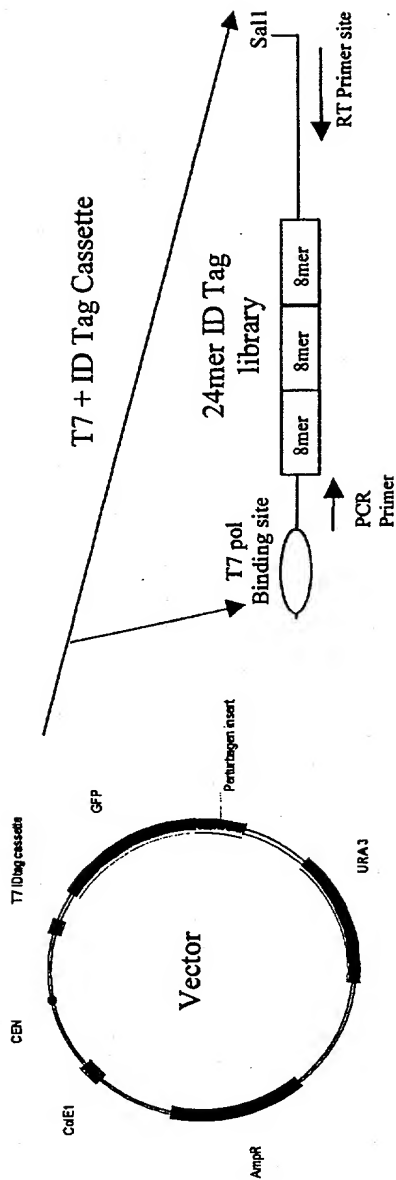
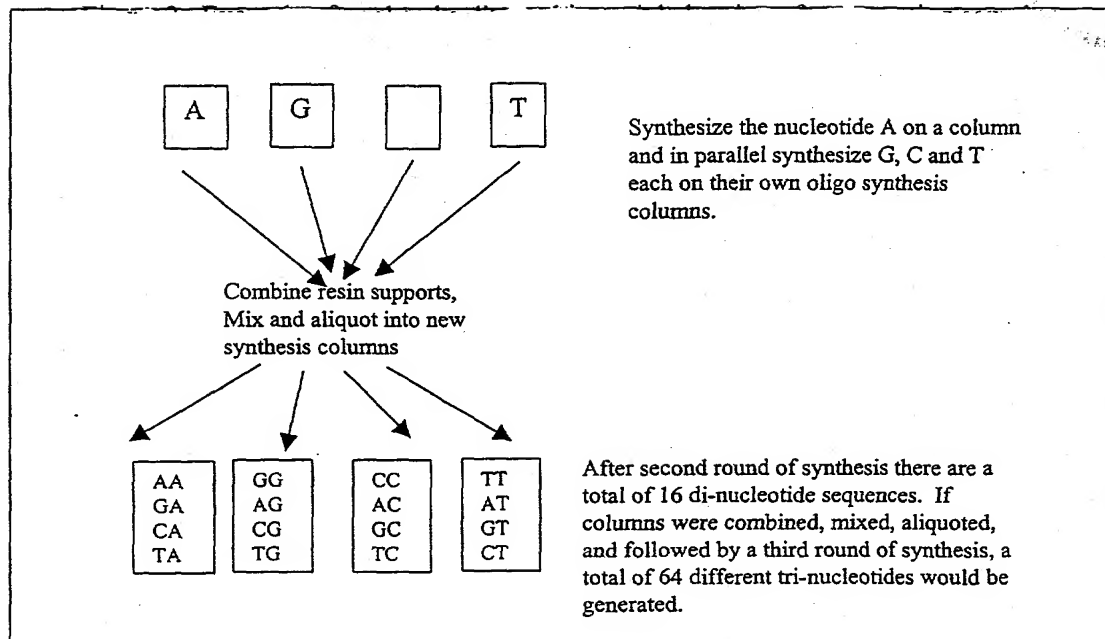


FIG. 20



**FIG. 21**

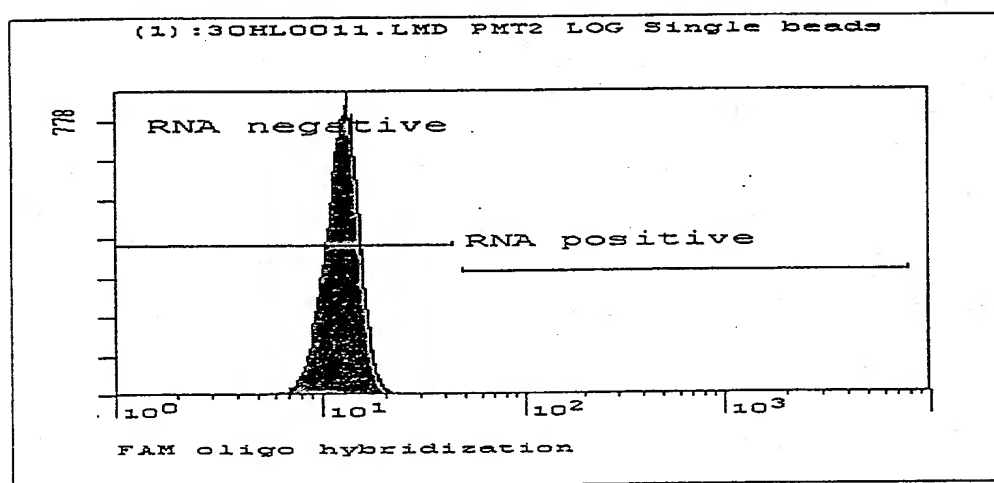
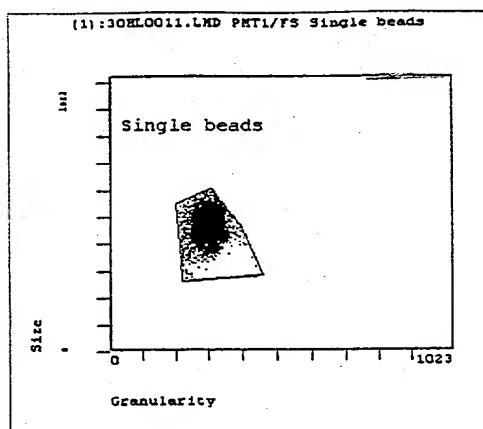


FIG. 23

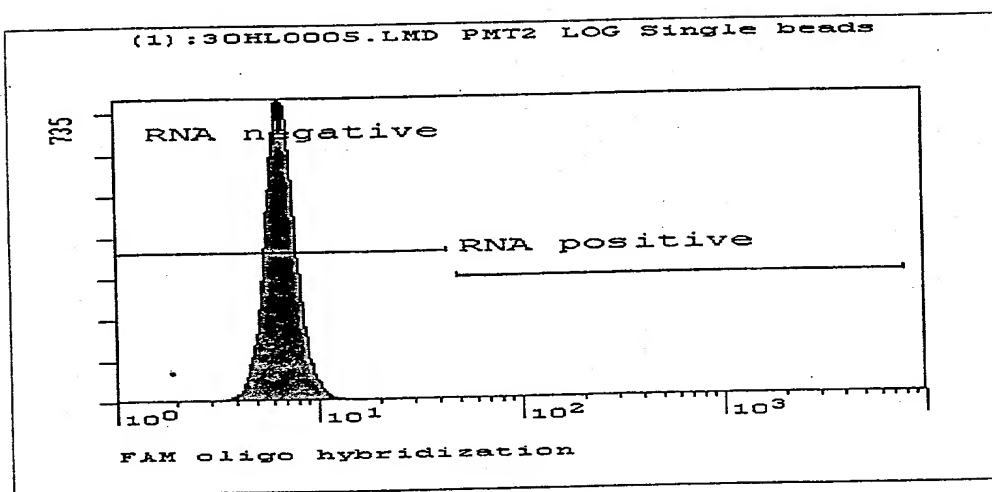
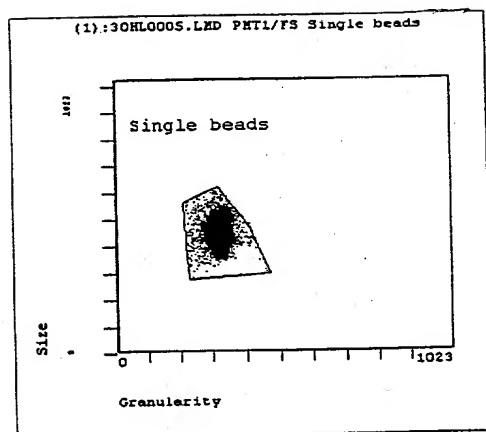


FIG. 24

FIG. 25

